Location Steps and Paths

Understanding Location Steps and Paths

How do location paths work? We took a look at location paths in the overview in Chapter 1, where we saw that location paths look much like directory paths. For example, you might store section one of chapter one of part one of a novel in a directory with this path if you’re using Windows:

\novel\part1\section1\chapter1

or this path if you’re using Unix:

/novel/part1/section1/chapter1

The idea behind XPath location paths is very much the same except that, as we’ll see in this chapter and the next, the XPath syntax can get much more complex and detailed than directory paths. Like directory paths, you build location paths out of individual steps, called location steps, separated by / or //. You use these steps to specify the data you want, just as you use successive directories in a file path to get to the location of a file.

For example, in the XPath location path

/library/book/title[2]/text()

we begin with a /, which matches the root node of the document, followed by, in order:

- The location step library, which matches the child <library> element of the root node
- The location step book, which matches the child <book> element of the <library> element
The location step title[2], which matches the second <title> element of the <book> element

text(), which matches the text in the <title> element

In fact, the XPath location path /library/book/title[2]/text() uses abbreviated XPath syntax, which we're going to see in this chapter. Here's the full version of this XPath location path, where the child element nature of each successive location step is spelled out:

/child::library/child::book/child::title[position()=2]/child::text

Okay—now let's get to the details.

The Parts of a Location Step

Each location step is made up of an axis, a node test, and zero or more predicates, like this (where the * symbol means “zero or more of”):

axis :: node-test [predicate]*

For example, in the location step child::name[position() = 2], child is the name of the axis (which you follow with ::), name is the node test and [position() = 2] is the predicate. In this case, the child axis selects child nodes of the node you start at, which is the context node. We're going to take a look at axes, node tests, and predicates in detail in this chapter. This location step can also be abbreviated, using the abbreviated syntax we'll see in this chapter, as name[2] (this works because the child axis is the default axis, and when you use position predicates like [position() = 2], XPath lets you abbreviate them as simply [2]).

Location paths can be made up of one or more location steps, such as child::name[position() = 2]/child::firstName, which selects all the <firstName> elements that are children of the second <name> child element of the context node. This location path can also be abbreviated as name[2]/firstName.

When you start the location path (not a location step) with / or //, the location path is called an absolute location path because you're specifying the path from the root node of the XML document (just as starting a file path with / in Unix starts the path from the root directory and makes the path an absolute one). Otherwise, the location path is relative, starting with the context node.

For example, take a look at this node tree again:

```
root
  |
  |
  element: <library>
```
Beginning a location path with / (or //, as we’ll see) makes that path start with the document’s root node, making this an absolute location path and making the root node the context node for the first location step in the path. To move to the child <library> element, you add the location step library like this: /library (using unabbreviated syntax, that would be /child::library). That makes the context node for the next location step the <library> element. To move to the <book> element starting from the <library> context node, you add book (or child::book) to the location path like this: /library/book and so on, all the way down to the text in the second <title> element, which you reach with /library/book/title[2]/text(). These are all absolute location paths, starting from the root node.

On the other hand, if you’ve got a context node already set—some XPath software lets you specify context nodes—your location paths can be relative. For example, if you’ve set the context node to the <library> element, the relative location path book/title[2]/text() will locate the text in the second <title> element in the document.

Now it’s time to get systematic about our study of location steps, which we’re going to do by taking a look at what kind of axes, node tests, and predicates XPath are available.

**XPath Axes**

We’re going to take a look at all 13 XPath 1.0 axes in this chapter. For example, in the location step child::planet, which refers to all <planet> elements that are children of the context node, child is the axis (and as you now know, you can also abbreviate this location step as planet, because child is the default axis). Here are all the XPath 1.0 axes:

- The ancestor axis holds the ancestors of the context node; the ancestors of the context node are the parent of context node and the parent’s parent and so forth, back to and including the root node.
- The ancestor-or-self axis holds the context node and the ancestors of the context node.
- The attribute axis holds the attributes of the context node.
- The child axis holds the children of the context node.
- The descendant axis holds the descendants of the context node. A descendant is a child or a child of a child and so on.
The descendant-or-self axis contains the context node and the descendants of the context node.

The following axis holds all nodes in the same document as the context node that come after the context node.

The following-sibling axis holds all the following siblings of the context node. A sibling is a node on the same level as the context node.

The namespace axis holds the namespace nodes of the context node.

The parent axis holds the parent of the context node.

The preceding axis contains all nodes that come before the context node.

The preceding-sibling axis contains all the preceding siblings of the context node. A sibling is a node on the same level as the context node.

The self axis contains the context node.

Each XPath location step must specify an axis (or use the default child axis), as in this location path we’ve already seen: /child::library/child::book/child::title[2]/child::text.

You can also see the various XPath axes listed in Table 3.1, which lists the support for these axes by XML processors in Internet Explorer. The XML processor in Internet Explorer is called MSXML; MSXML 2.0 was the version in Internet Explorer 5.0, early versions of Internet Explorer 6.0 used MSXML3, and the current version is MSXML4. If you’re using the .NET platform, your version of MSXML is MSXML.NET.

<table>
<thead>
<tr>
<th>AXIS</th>
<th>ABBREVIATION</th>
<th>MSXML2</th>
<th>MSXML3</th>
<th>MSXML4</th>
<th>MSXML.NET</th>
</tr>
</thead>
<tbody>
<tr>
<td>ancestor</td>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>ancestor-or-self</td>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>attribute</td>
<td>@</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>child</td>
<td>(default)</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>descendant</td>
<td>//</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>descendant-or-self</td>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>following</td>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>following-sibling</td>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>namespace</td>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>parent</td>
<td>..</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>preceding</td>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>preceding-sibling</td>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>self</td>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>
**XPath Node Tests**

When you use an axis in a location step, you're telling XPath where to look and identifying a set of nodes. A node test tells XPath which of the nodes in that set you're interested in.

There are a number of ways to create node tests. You can use names of nodes as node tests, or the wildcard * to select element or attribute nodes (note especially that * matches only elements and attributes, not just any kind of node). For example, the location step `child::*/child::name` selects all `<name>` elements that are grandchildren of the context node. To match attributes, you'd use the attribute axis like this: `attribute::*`. Besides node names and the wildcard character, you can also use these node tests in XPath 1.0:

- The `*` wildcard character matches any element or attribute name.
- A name matches a node with that name (for example, `planet` will match a `<planet>` element).
- The `comment()` node test selects comment nodes.
- The `node()` node test selects any type of node.
- The `processing-instruction()` node test selects a processing instruction node. You can specify the name of the processing instruction to select in the parentheses.
- The `text()` node test selects a text node.

You can see the XPath 1.0 node tests in Table 3.2, along with the XML processor version that supports them in the Internet Explorer.

<table>
<thead>
<tr>
<th>AXIS</th>
<th>MSXML2</th>
<th>MSXML3</th>
<th>MSXML4</th>
<th>MSXML.NET</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>name</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>comment()</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>node()</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>processing-instruction()</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>text()</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>

The node test lets you specify what nodes you want to work with in an XPath location step. For example, take a look at our sample node tree:

```
root
  |
  |
  element: <library>
```
You can start at the root node with /, and then use the child axis and the node test library to move to the <library> element—giving you the location path /child::library. You can see this at work in the XPath Visualiser in Figure 3.1.

And you can move to the <book> element under the <library> element with another location step involving this child axis and a node test like this: /child:library/child::book. You can see what this looks like in the XPath Visualiser in Figure 3.2.
The next level down in the node tree holds two <title> elements, however. What if we only want to work with the second one? If we used the location path
/child::library/child::book/child::title, we’d match both <title> elements, so we need more than a node test here—we need to use a predicate.

**XPath Predicates**

The next part of a location step, which follows the node text, is the predicate. A location step doesn’t need a predicate, but if you use a predicate, you can specify even more about the node or nodes you want to match.

You often use one of the built-in XPath functions in predicates. For example, take a look at the location step child::planet[position() = 2]. In this case, the predicate, which is always enclosed between [ and ], is position() = 2. This means that the value the built-in XPath function position() returns must indicate that this is the second <planet> child in order for the location step to match (this location step can also be abbreviated as planet[2]). In this way, this predicate narrows down the search from the node-set of all <planet> children of the context node down to the second <planet> child.

Now we’re in a position to select the second <title> element in our XML document that has this node tree:

```
root
  |
  element: <library>
    |
    element: <book>
      |
      element: <title>
        |
        text: "I Love XPath"
        |
        element: <title>
          |
          text: "XPath is the BEST"
```

To move to the second <title> element, we can use the location path
/child::library/child::book/child::title[2], as you see in the XPath Visualiser in Figure 3.3 (this location path can also be abbreviated as /library/book/title[2]).

As you can see, expressions in predicates can let you narrow down the search from a whole node-set to just the nodes you’re looking for.

Here’s another node-test example—to select the <title> element that contains the text “I Love XPath”, you can use this XPath location path with the text() node test:
/child::library/child::book/child::title[text()="I Love XPath"], as you see in Figure 3.4.
CHAPTER 3  Location Steps and Paths

Now we're able to construct XPath location paths like `/child::library/child::book/child::title[2]` using axes, node tests, and predicates. Each part of a location step, the axis, node test, and predicate, narrows down the set of nodes you're working with.

You can also build up XPath expressions much as you can location paths. For example, the `text()` node test will return the text in a node, so if you want to extract the text of the `<title>` element, you can use the XPath expression `/child::library/child::book/child::title[2]/text()`, which evaluates not to a node-set, but to a text string, “XPath is the BEST”, as you can see in Figure 3.5.

You can use multiple predicates in the same location step—for example, say that we added a `language` attribute to each `<planet>` element in our planetary data XML document, as you see in `ch03_01.xml` (see Listing 3.1).
LISTING 3.1 Adding a Language Attribute (ch03_01.xml)

```xml
<?xml version="1.0" encoding="utf-8"?>
<planets>

<planet language="English">
    <name>Mercury</name>
    <mass units="(Earth = 1)">.0553</mass>
    <day units="days">58.65</day>
    <radius units="miles">1516</radius>
    <density units="(Earth = 1)">.983</density>
    <distance units="million miles">43.4</distance><!--At perihelion-->
</planet>

<planet language="English">
    <name>Venus</name>
    <mass units="(Earth = 1)">.815</mass>
    <day units="days">116.75</day>
    <radius units="miles">3716</radius>
    <density units="(Earth = 1)">.943</density>
    <distance units="million miles">66.8</distance><!--At perihelion-->
</planet>

<planet language="English">
    <name>Earth</name>
    <mass units="(Earth = 1)">1</mass>
    <day units="days">1</day>
</planet>
</planets>
```
LISTING 3.1  Continued

```xml
<radius units="miles">2107</radius>
<density units="(Earth = 1)">1</density>
<distance units="million miles">128.4</distance> <!-- At perihelion -->
</planet>
</planets>
```

What if we wanted to reach the second `<planet>` element that has a `language` attribute set to “English”? We could do that with a location path like this:

```
/planets/planet[attribute::language = "English"][position() = 2],
```

as you see in Figure 3.6. In this way, you can handle multiple conditions with multiple predicates.

**FIGURE 3.6**  Using `/planets/planet[attribute::language = "English"][position() = 2].`

Some Examples of XPath Location Paths

There’s nothing like seeing all this at work to understand what’s going on, so here are a number of location path examples:

- **child::planet**—Returns the `<planet>` element children of the context node.

- **child::***—Returns all element children (* only matches elements, or attributes if you use it with the attribute axis) of the context node.

- **child::text()**—Returns all text node children of the context node.
- `child::node()`—Returns all the children of the context node, no matter what their node type is.
- `attribute::units`—Returns the `units` attribute of the context node.
- `descendant::planet`—Returns the `<planet>` element descendants of the context node.
- `ancestor::planet`—Returns all `<planet>` ancestors of the context node.
- `ancestor-or-self::planet`—Returns the `<planet>` ancestors of the context node. If the context node is a `<planet>` as well, also returns the context node.
- `descendant-or-self::planet`—Returns the `<planet>` element descendants of the context node. If the context node is a `<planet>` as well, also returns the context node.
- `self::planet`—Returns the context node if it is a `<planet>` element.
- `child::name/descendant::planet`—Returns the `<planet>` element descendants of the child `<name>` elements of the context node.
- `child::*/child::planet`—Returns all `<planet>` grandchildren of the context node.
- `/`—Returns the root node (that is, the parent of the document element).
- `/descendant::planet`—Returns all the `<planet>` elements in the document.
- `/descendant::planet/child::name`—Returns all the `<name>` elements that have a `<planet>` parent.
- `child::planet[position() = 3]`—Returns the third `<planet>` child of the context node.
- `child::planet[position() = last()]`—Returns the last `<planet>` child of the context node.
- `child::planets/child::planet[position() = 4 ]/child::name[position() = 3]`—Returns the third `<name>` element of the fourth `<planet>` element of the `<planets>` element.
- `child::planet[position() > 3]`—Returns last() location path > last() all the `<planet>` children of the context node after the first three.
- `preceding-sibling::name[position() = 2]`—Returns the second previous `<name>` sibling element of the context node.
- `child::planet[attribute::color = "RED"]`—Returns all `<planet>` children of the context node that have a `color` attribute with value of “RED”.

Understanding Location Steps and Paths
■ child::planet[[attribute::color = "RED"]][position() = 3]—Returns the third <planet> child of the context node that has a color attribute with value of “RED”.

■ child::planet[position() = 3][attribute::color="RED"]—Returns the third <planet> child of the context node, only if that child has a color attribute with value of “RED”.

■ child::planet[child::name]—Returns the <planet> children of the context node that have <name> children.

As you can see, some of this syntax is pretty involved, and a little lengthy to type. However, there is an abbreviated form of XPath syntax, and we’ll look at that next.

Using XPath Abbreviated Syntax

There are a number of abbreviations you can take advantage of in XPath syntax. Here are the rules:

■ self::node() can be abbreviated as .

■ parent::node() can be abbreviated as ..

■ child::nodename can be abbreviated as nodename

■ attribute::nodename can be abbreviated as @nodename

■ /descendant-or-self::node()// can be abbreviated as //

You can also abbreviate predicate expressions like [position() = 3] as [3]. Using the abbreviated syntax makes XPath expressions a lot easier to write. For example, attribute::units can be abbreviated as @units, you can refer to the context node itself as simply ., and you can refer to the current node and any descendants as //.

The // syntax in particular is useful and important. Take a look at ch03_01.xml, for example, the XML document where we're storing planetary data. In that XML document, we have three <planet> elements as children of the main <planets> element:

```xml
<planets>
  <planet language="English">
    <name>Mercury</name>
    <mass units="(Earth = 1)">.0553</mass>
    <day units="days">58.65</day>
  </planet>
  ...
</planets>
```
To select all three <planet> elements, you can use the absolute XPath expression /planets/planet, which starts at the XML document’s root node, finds the <planets> element, and then matches the three <planet> child elements. That's fine if you know exactly where in the XML document the elements you want are and so can specify a direct path to them.

But you can also use //planet to select all three <planet> elements, because //planet will find the <planet> elements by checking the root node and all descendants for <planet> elements. That's the power of //—when you want to search for nodes that may be anywhere in a document, use //.

How about some examples of location paths using abbreviated syntax? Here are a number of examples:

- planet—Returns the <planet> element children of the context node.
- *—Returns all element children of the context node.
- text()—Returns all text node children of the context node.
- @units—Returns the units attribute of the context node.
- @*—Returns all the attributes of the context node.
- planet[3]—Returns the third <planet> child of the context node.
- planet[first()]—Returns the first <planet> child of the context node.
- */planet—Returns all <planet> grandchildren of the context node.
- /planets/planet[3]/name[2]—Returns the second <name> element of the third <planet> element of the <planets> element.
Using the XPath Axes

There are 13 axes to master, and we'll take a look at them here, complete with examples. To understand how something like XPath works, there's no better way than seeing it at work as much as possible.

We'll take a look at various examples using XPath Visualiser, and we'll also take a look at some examples using the XPath axes with XSLT. You don't really have to understand the XSLT at this point—you can just pick out the XPath expression inside the example. But XSLT is important when working with XPath, as we're going to see in Chapter 5, and here it will help us out when XPath Visualiser can't (as with the namespace axis, which XPath Visualiser doesn't display visually). We're already familiar with the child and attribute axes, so we won't introduce them here, but we will introduce all the other axes now, beginning with the ancestor axis.

Using the ancestor Axis

The ancestor axis contains all the ancestors of the context node, including its parents, grandparents, great-grandparents, and so on. This axis always contains the root node (unless the context node is the root node).

Here's an example using XPath Visualiser. In this case, we'll use the location path //planet/day to select the <day> elements in our planetary data example, ch03_01.xml. Then we'll work backward with the ancestor axis to find the <planet> ancestor of each <day> element like this: //planet/day/ancestor::planet. You can see the results in Figure 3.7 (note that we're only searching for <planet> ancestors with this location path, so only <planet> ancestors are selected).
Using the XPath Axes

Here’s an example doing the same thing using XSLT. As discussed in Chapter 1, in XSLT you create a template with an `<xsl:template>` element to match nodes. In this case, we want to match `<day>` elements:

```xml
<xsl:template match="day">
    
    
</xsl:template>
```

Now we’ll use an `<xsl:for-each>` element to loop over all ancestors of the `<day>` element, using the XPath ancestor axis:

```xml
<xsl:template match="day">
    <xsl:for-each select="ancestor::*">
        
        
    </xsl:for-each>
</xsl:template>
```

To display the name of the ancestor element, we can use the XSLT `<xsl:value-of>` element. We can extract the name of the current planet with the XPath expression `./name`, where `.` selects the context node. Here’s what that looks like in XSLT:

```xml
<?xml version='1.0' encoding='utf-8'?>
<xsl:stylesheet version='1.1'
xmlns:xsl='http://www.w3.org/1999/XSL/Transform'>
<xsl:output method="xml"/>
```
<xsl:template match="day">
  <xsl:for-each select="ancestor::*">
    <xsl:value-of select="./name"/>
  </xsl:for-each>
</xsl:template>

<xsl:template match="planet">
  <xsl:apply-templates select="day"/>
</xsl:template>
</xsl:stylesheet>

And here’s the result when you use this stylesheet on ch03_01.xml—as you can see, we’ve been able to pick out the names of the ancestors of the <day> elements in our document:

```xml
<?xml version="1.0" encoding="utf-8"?>
Mercury
Venus
Earth
```

### Using the **ancestor-or-self** Axis

The **ancestor-or-self** axis contains all the ancestors of the context node, and the context node itself. That means, among other things, that this axis always contains the root node.

Here’s an example using XPath Visualiser. In this case, we’ll use this axis to select all ancestors of <day> elements, as well as the <day> element itself this way: `/planet/day/ancestor-or-self::*`. You can see the results in Figure 3.8.

**FIGURE 3.8** Using the **ancestor-or-self** axis.
Here's an example using XSLT and the ancestor-or-self axis. In this case, we're going to add author attributes set to “Thaddeus” throughout our document like this:

```xml
<?xml version='1.0' encoding='utf-8'?>
<?xml-stylesheet type='text/xml' href='planets.xsl'?>
<planets author='Thaddeus'>
  <planet author='Thaddeus' language='English'>
    <name>Mercury</name>
    <mass units='(Earth = 1)'>0.0553</mass>
    <day units='days'>58.65</day>
    <radius units='miles'>1516</radius>
    <density units='(Earth = 1)'>0.983</density>
    <distance units='million miles'>43.4</distance> <!-- At perihelion -->
  </planet>
  <planet author='Thaddeus' language='English'>
    <name)Venus</name>
    <mass units='(Earth = 1)'>0.815</mass>
    <day units='days'>116.75</day>
    <radius units='miles'>3716</radius>
    <density units='(Earth = 1)'>0.943</density>
    <distance units='million miles'>66.8</distance> <!-- At perihelion -->
  </planet>
  <planet language='English'>
    <name>Earth</name>
    <mass units='(Earth = 1)'>1</mass>
    <day units='days'>1</day>
    <radius units='miles'>2107</radius>
    <density units='(Earth = 1)'>1</density>
    <distance units='million miles'>128.4</distance> <!-- At perihelion -->
  </planet>
</planets>
```

Now say that you want to list by name all ancestors of `<day>` elements that have an `author` attribute—as well as the current `<day>` element if it has an `author` attribute. To do that, you can use the XPath location path `ancestor-or-self::*[@author]`, which matches all nodes and ancestors that have an `author` attribute. Here's what it looks like in XSLT:

```xml
<?xml version='1.0' encoding='utf-8'?>
<xsl:stylesheet version='1.1'
  xmlns:xsl='http://www.w3.org/1999/XSL/Transform'>
  <xsl:output method='xml'/>
</xsl:stylesheet>
```
CHAPTER 3  Location Steps and Paths

<?xml version="1.0" encoding="UTF-8"?>
<br>planets  planet  day
<br>planets  planet
<br>planets

Using the descendant Axis

The descendant axis contains all the descendants of the context node. Note that this does not include any attributes or namespace nodes.

Here's an example using XPath Visualiser. In this case, we'll select all descendants of <planet> elements with the location path //planet/descendant::*, as you see in Figure 3.9.

FIGURE 3.9  Using the descendant axis.
Here's an example using XSLT. In this case, we'll check a document to see if it includes a `<planet>` element for Mercury, and if so, we'll include this element in the result:

```xml
<info>Sorry, Mercury cannot be found at this time.</info>
```

To match Mercury's `<planet>` element, all you have to do is to check whether any text node descendant of a `<planet>` element holds the string "Mercury" this way:

```xml
<?xml version='1.0' encoding='utf-8'?>
<xsl:stylesheet version='1.0'
    xmlns:xsl='http://www.w3.org/1999/XSL/Transform'>
    <xsl:output method='xml'/>
    <xsl:template match='planet[descendant::text()='Mercury']'>
        <info>Sorry, Mercury cannot be found at this time.</info>
    </xsl:template>
    <xsl:template match='*'>
        <xsl:apply-templates select='*' />
    </xsl:template>
</xsl:stylesheet>
```

That's all it takes. Here's the result, showing the `<info>` element:

```xml
<?xml version='1.0' encoding='utf-8'?>
<info>Sorry, Mercury cannot be found.</info>
```

**Using the descendant-or-self Axis**

The descendant-or-self axis contains all the descendants of the context node, and the context node itself. Note, however, that it does not contain any attributes or namespace nodes.

You can see an example in Figure 3.10, where we’re selecting all `<planet>` elements and their descendants with the XPath location path `//planet/descendant-or-self::*`.

Here's an example doing the same thing using XSLT. In this case, we'll use an XSLT template to match all `<planet>` elements and then loop over all nodes in the node-set returned by using the descendant-or-self axis, displaying each node's name:

```xml
<?xml version='1.0' encoding='utf-8'?>
<xsl:stylesheet version='1.0'
    xmlns:xsl='http://www.w3.org/1999/XSL/Transform'>
    <xsl:output method='xml'/>
    <xsl:template match='planet[descendant-or-self::*]'>
        <xsl:apply-templates select='.' />
    </xsl:template>
</xsl:stylesheet>
```
CHAPTER 3 Location Steps and Paths

<?xml version="1.0" encoding="UTF-8"?>

planet name mass day radius density distance

Using the following Axis

The following axis contains all nodes that come after the context node in document order, excluding any of the context node's descendants—and also excluding attribute nodes and namespace nodes.

You can see an example in the XPath Visualiser in Figure 3.11, where we’re using this axis to select the following elements after the <mass> element in the first <planet> element, using the XPath location path /planets/planet[1]/mass/following::*.

Figure 3.10 Using the descendant-or-self axis.

That’s all it takes. Here’s the result, where we’ve been able to list the name of all the descendants of <planet> elements, as well as the <planet> elements themselves, using the descendant-or-self axis:

<?xml version="1.0" encoding="UTF-8"?>

planet name mass day radius density distance
planet name mass day radius density distance
planet name mass day radius density distance

Using the following Axis

The following axis contains all nodes that come after the context node in document order, excluding any of the context node's descendants—and also excluding attribute nodes and namespace nodes.

You can see an example in the XPath Visualiser in Figure 3.11, where we’re using this axis to select the following elements after the <mass> element in the first <planet> element, using the XPath location path /planets/planet[1]/mass/following::*.
Here's an example using XSLT to do the same thing. In this case, we're matching the first `<planet>` element in an XSLT template and displaying the names of the following elements:

```xml
<?xml version="1.0" encoding="utf-8"?>
<xsl:stylesheet version="1.1"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="xml"/>

<xsl:template match="planet[1]">
  <xsl:for-each select="mass/following::*">
    <xsl:value-of select="local-name()"/>
    <xsl:text> </xsl:text>
  </xsl:for-each>
</xsl:template>

<xsl:template match="*">
  <xsl:apply-templates select="*">
</xsl:template>

</xsl:stylesheet>
```

Here's what the result looks like. Note that we've been able to get all the elements following the `<mass>` element in the first `<planet>` element, and then all the following elements in the rest of the document:
Using the following-sibling Axis

The following-sibling axis contains all the following siblings of the context node. You can see an example in the XPath Visualiser in Figure 3.12, where we're using the XPath location path /planets/planet[1]/mass/following-sibling::* to select all following sibling nodes of the <mass> element in the first <planet> element.

Here's how this example works in XSLT; in this case, we're also matching the first <planet> element's <mass> element and then getting its following sibling elements:

```xml
<?xml version="1.0" encoding="utf-8"?>
<xsl:stylesheet version="1.1"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:output method="xml"/>

    <xsl:template match="planet[1]">
        <xsl:for-each select="mass/following-sibling::*">
            <xsl:value-of select="local-name()"/>
            <xsl:text> </xsl:text>
        </xsl:for-each>
    </xsl:template>
</xsl:stylesheet>
```
Here's the result—as you can see, we've caught all the siblings following the <mass> element in the first <planet> element:

```xml
<?xml version="1.0" encoding="UTF-8"?>
day radius density distance

Using the namespace Axis

The namespace axis contains the namespace nodes of the context node—note that the axis will be empty unless the context node is an element. An element will have a namespace node for

- Every attribute of the element whose name starts with “xmlns:”.
- Every attribute of an ancestor element whose name starts “xmlns:” (unless, of course, the element itself or a nearer ancestor redeclares the namespace).
- An xmlns attribute, if the element, or some ancestor, has an xmlns attribute.

XPath Visualiser doesn’t handle this axis visually, so we’ll rely on XSLT here. Here, we’ll add an XML namespace declaration to the <planets> element, using the namespace “http://www.XPathCorp.com” like this:

```xml
<?xml version="1.0" encoding="utf-8"?>
<?xml-stylesheet type="text/xml" href="planets.xsl"?>
<planets xmlns="http://www.XPathCorp.com">
  <planet>
    <name>Mercury</name>
    <mass units="(Earth = 1)">.0553</mass>
    <day units="days">58.65</day>
    <radius units="miles">1516</radius>
    <density units="(Earth = 1)">.983</density>
    <distance units="million miles">43.4</distance><!--At perihelion-->
  </planet>

```
In XSLT, we can check the namespaces used in the `<planets>` element like this:

```xml
<?xml version="1.0" encoding="utf-8"?>
<xsl:stylesheet version="1.1"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:output method="xml"/>

    <xsl:template match="planets">
        <xsl:value-of select="namespace::*"/>
    </xsl:template>

</xsl:stylesheet>
```

And here’s the result, showing that we can indeed pick out the namespace:

```xml
<?xml version="1.0" encoding="UTF-8"?>
http://www.XPathCorp.com
```

**Using the parent Axis**

The parent axis contains the parent (and only the parent) of the context node, if there is one.

You can see an example in XPath Visualiser in Figure 3.13. Here, we’re picking out the parent elements of all `<day>` elements with the XPath location path `//day/parent::*`. 

![Figure 3.13](image)

**FIGURE 3.13** Using the parent axis to extract data.
And here’s the same example in XPath. In this case, we’ll match all `<day>` elements and get the names of their parent elements. Here’s what it looks like:

```xml
<?xml version="1.0" encoding="utf-8"?>
<xsl:stylesheet version="1.1"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="xml"/>

<xsl:template match="//day">
  <xsl:for-each select="parent::*">
    <xsl:value-of select="local-name()"/>
    <xsl:text> </xsl:text>
  </xsl:for-each>
</xsl:template>

<xsl:template match="*">
  <xsl:apply-templates select="*"/>
</xsl:template>
</xsl:stylesheet>
```

And here’s the result:

```xml
<?xml version="1.0" encoding="UTF-8"?>
planet planet planet
```

**Using the preceding Axis**

The preceding axis contains all nodes that are before the context node in document order, excluding any ancestors of the context node, and also excluding attribute nodes and namespace nodes.

Here’s an example using XPath Visualiser. In this case, we’ll select all elements preceding the `<density>` element in the first planet element with the XPath location path `//planet[1]/density/preceding::*`, as you can see in Figure 3.14.

Let’s give this axis a try in XSLT. In this case, say that we want to set the content of the `<distance>` element to the text “This planet is farther than Mercury from the sun.” if the current planet is indeed farther from the sun than Mercury. One way to do that is to see if Mercury comes before the current planet in document order, using the preceding axis:

```xml
<?xml version="1.0" encoding="utf-8"?>
<xsl:stylesheet version="1.1"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="xml"/>
```
CHAPTER 3  Location Steps and Paths

If the current planet does come after Mercury, this example inserts the message in its <distance> element, as you see in this result:

```xml
<?xml version="1.0" encoding="utf-8"?>
<?xml-stylesheet type="text/xml" href="planets.xsl"?>
<planets>
  <planet>
    <name>Mercury</name>
    <mass units="(Earth = 1)">0.0553</mass>
    <day units="days">58.65</day>
    <radius units="miles">1516</radius>
    <density units="(Earth = 1)">0.983</density>
    <distance units="million miles">43.4</distance>
    <!--At perihelion-->
  </planet>
</planets>
```

FIGURE 3.14  Using the preceding axis to select elements.
Using the preceding-sibling Axis

The preceding-sibling axis contains all the preceding siblings of the context node. Note that if the context node is an attribute node or namespace node, the preceding-sibling axis won’t hold anything.

You can see an example in the XPath Visualiser in Figure 3.15, where we’re using the XPath location path //planet[2]/preceding-sibling::* to select all preceding siblings of the second <planet> element. Note that just the first <planet> element is selected. On the other hand, if we had used //planet[2]/preceding::*, not only would the first <planet> element be selected, but all that element’s child elements would be selected as well.

Here’s a more advanced example using XSLT. In this case, we’ll replace the <distance> element in Mercury’s <planet> element with <distance>This planet is the closest to the sun.</distance>. If we’re matching <distance> elements, how can we make sure that we’ve got Mercury’s <distance> element? We can check the current <distance> element’s preceding siblings and look for the text “Mercury”. Here’s what it looks like in XSLT:

```xml
<?xml version='1.0'?
<xsl:stylesheet version='1.0'
xmlns:xsl='http://www.w3.org/1999/XSL/Transform'>
<xsl:output method='xml'/>
```
CHAPTER 3  Location Steps and Paths

FIGURE 3.15  Using the preceding-sibling axis to extract data.

And here's the result:

```xml
<?xml version="1.0" encoding="utf-8"?>
<?xml-stylesheet type="text/xml" href="planets.xsl"?>
<planets>
  <planet language="English">
    <name>Mercury</name>
    <mass units="(Earth = 1)">.0553</mass>
    <day units="days">58.65</day>
    <radius units="miles">1516</radius>
    <density units="(Earth = 1)">.983</density>
    <distance>This planet is the closest to the sun.</distance>
  </planet>
</planets>
```
Using the self Axis

The self axis contains just the context node, and you can abbreviate "self::node()" as ".". This is a useful axis to know about, because as you know, if you omit the axis, the default is child::, but sometimes you want to refer to the current node instead. For example, [self::planet] is true only if the context node is a <planet> element.

You can see an example using this axis in XPath Visualiser in Figure 3.16, where we're using the XPath location path //*[self::radius] to select <radius> elements in ch03_01.xml (this location path is equivalent to //radius).

Here's an example using XSLT. In this case, we'll use one template to match both <name> and <day> elements in the same template. We can do that by matching name | day in a template like this (more on how this works in the next section):

```xml
<xsl:template match="name | day">
  .
  .
  .
</xsl:template>
```

At this point, we've matched both <name> and <day> elements—but suppose that in the body of the template we actually want to treat these elements differently. To do that, we have to check if we're dealing with a <name> element or a <day> element, which we can do with the
XSLT element `<xsl:if>`, where you assign the condition to test to this element’s `test` attribute. Here’s what it looks like in XSLT:

```
<xsl:template match="name | day">
  <xsl:if test="self::name">
    <xsl:value-of select="."/>
  </xsl:if>
  <xsl:if test="self::day">
    <xsl:value-of select="."/>
    <xsl:text> </xsl:text>
    <xsl:value-of select="@units"/>
  </xsl:if>
</xsl:template>
```

FIGURE 3.16 Using the `self` axis to select `<radius>` elements.

So now we’ve taken a look at all 13 axes, from the ancestor axis to the self axis. Note that you can combine location paths with the `|` operator—we’ll take a closer look at that now.

Creating Compound Location Paths

In XPath, you can combine location paths with the `|`, or “pipe” character. We’ve already seen that at work like this:
You can use any kind of location paths with the pipe character, like this:

```
<xsl:template match="name/firstName | day/author/lastName">
  
  
</xsl:template>
```

When you use a compound location path, each path is evaluated separately—that is, the context node for the first path doesn’t affect the context node for the other path. And you’re not restricted to two paths; you can connect as many as you like with pipes:

```
<xsl:template match="name/firstName | day/author/lastName | distance | mass/kilograms">
  
  
</xsl:template>
```

When it comes to predicates, however, the situation is different. In XPath, you combine conditions in a predicate with the and and or operators. For example, if you wanted to match all `<distance>` elements that have `units` attributes set to “million miles” and `language` attributes set to “English”, you could use the `and` operator like this:

```
<xsl:template match="distance[@units='million miles' and @language='English']">
  
  <distance>This is the planet Mercury, closest to the Sun.</distance>
</xsl:template>
```

```
<xsl:template match="distance[@units='million miles' and @language='English']">
  
  <distance>This is the planet Mercury, closest to the Sun.</distance>
</xsl:template>
```

**USING QUOTATION MARKS**

Notice the quotation marks here—to avoid confusing the XSLT processor, we’re using single quotes to surround the attribute values here, because the whole XPath expression is surrounded in double quotes. You could also use the `or` operator if you wanted to match `<distance>` elements that have `units` attributes set to “million miles” or `language` attributes set to “English”, like this:

```
<xsl:template match="distance[@units='million miles' and @language='English']">
  
  <distance>This is the planet Mercury, closest to the Sun.</distance>
</xsl:template>
```
Here's another example: child::*[self::name or self::mass], which returns both the <name> and <mass> children of the context node.

You can connect as many conditions in a predicate as you want using and and or. And if it gets confusing, you can use parentheses like this:

```xml
<xsl:template match="author[(@firstName='Cary') and (@lastName='Grant')]">
  <distance>This is the planet Mercury, closest to the Sun.</distance>
</xsl:template>
```

Note also that compound location paths can also have predicates, of course, including compound predicates.

**Nesting Predicates**

It's also legal to nest predicates. Here's how you might select all <project> elements that have <name> descendants that in turn have a preceding sibling <active> element:

```xml
//project[descendant::name[preceding-sibling::active]]
```

You can also nest predicates when using compound location paths and even compound predicates.

**Using Predicates Without Operators**

The predicates we've used so far usually use operators, as in the location step `author[position()=4]`, where we're using the `=` operator (we'll see all the available operators in the next chapter). However, you don't need to use operators at all in predicates if you just want to test for the existence of a node.

For example, if you want to find all <notation> elements that contain at least one <author> element, you can use this location path: `//notation[descendant::author]`.

That's it for this chapter on location steps and location paths. We saw how XPath works with the data model introduced in Chapter 2 by using location paths, which are made up of location steps. Each location step, in turn, is made up of an axis, a node test, and a predicate. In Chapter 4, we're going to get more details on creating location paths as we see the operators and functions you can use in XPath predicates.
In Brief

- Each location step is made up of an axis, a node test, and zero or more predicates.
- An axis indicates how to search for nodes. Here are the XPath 1.0 axes:
  - The child axis
  - The attribute axis
  - The ancestor axis
  - The ancestor-or-self axis
  - The descendant axis
  - The descendant-or-self axis
  - The following axis
  - The following-sibling axis
  - The namespace axis
  - The parent axis
  - The preceding axis
  - The preceding-sibling axis
  - The self axis
- You can use these node tests in XPath 1.0:
  - The * wildcard character matches any element or attribute name.
  - A name matches a node with that name (for example, planet will match a <planet> element).
  - The comment() node test selects comment nodes.
  - The node() node test selects any type of node.
  - The processing-instruction() node test selects a processing instruction node.
  - The text() node test selects a text node.
- Predicates are enclosed in [ and ] and may contain any valid XPath 1.0 expression.